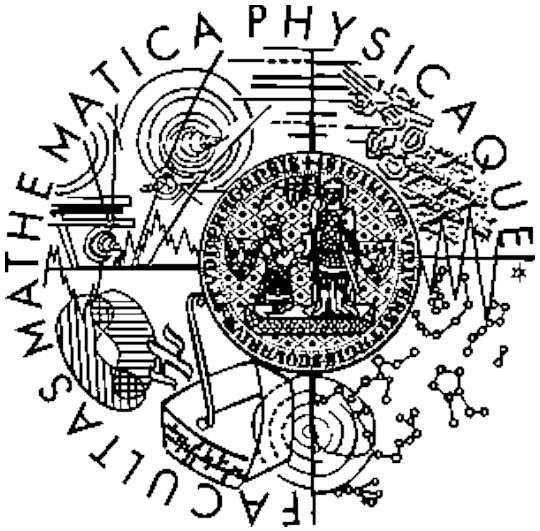
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Lukáš Tomasy

Munis Instant

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. David Hoksza Ph.D.

Studijní program: Informatika, programování

2011

Ďakujem vedúcemu bakalárskej práce za jeho odbornú pomoc pri tvorbe práce. Zároveň ďakujem pánovi Mgr. Tomášovi Lechnerovi za dovolenie použiť v tejto práci kód patriaci spoločnosti Triada spol. s r.o.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a len s použitím citovaných zdrojov. Súhlasím so zapožičiavaním práce.

V Prahe dňa 25. května 2011

Lukáš Tomasy
Názov práce: Munis Instant

Autor: Lukáš Tomasy

Katedra (ústav): Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. David Hoksza Ph.D.

e-mail vedúceho: hoksza@ksi.mff.cuni.cz

Abstrakt: Rýchle vyhľadávanie informácií bolo pre človeka vždy veľmi dôležité. V dnešnej dobe internetu a informačných systémov je azda ešte dôležitejšie. Snaha softwarových inžinierov preto smeruje k vývoju čo najsofistikovanejších systémov pre vyhľadávanie informácií. Projekt Munis Instant sa má pripojiť k tejto snahe a poskytnúť inteligentné vyhľadávanie na základe našepkávania relevantných dát z databázy informačného systému. Program je implementovaný ako knižnica programovaná na platforme .NET a dá sa použiť na vyhľadávanie v ľubovoľnej databáze podporovanej platformou .NET natívne (OleDbProvider). Snaha bola, aby nebol tento projekt viazaný na jeden konkrétny informačný systém.

Kľúčové slová: Vyhľadávanie, informačný systém, .NET

Title: Munis Instant

Author: Lukáš Tomasy

Department: Department of Software Engineering

Supervisor: RNDr. David Hoksza Supervisor's

e-mail address: hoksza@ksi.mff.cuni.cz

Abstract: Quick searching of information was always very important for people. Today, in era of internet and information system it is perhaps even more important. Therefore software engineers around the globe try to develop as sophisticated search system as possible. Project Munis Instant should join this initiative and should provide intelligent searching of informations based on instantly displaying relevant data from the database alongside with typing. Program is being developed as a library written in .NET platform and can be used over any database supported by .NET (OldDbProvider). The main goal was, the should be independent on database system or information system.

Keywords: Searching, information system, .NET

Obsah

1 Úvod

1.1 Doba informácií

Žijeme v dobe množstva informácií a poznatkov. Každým dňom je ich viac a viac. Bežný smrteľník už dávno nedokáže absorbovať také množstvo informácií aké ku svojej práci potrebuje. Potreba ukladať informácie je stará ako ľudstvo samotné. Už v praveku si ľudia dôležité momenty značili na steny jaskyne, neskôr používali kamenné a hlinené dosky. V staroveku ich nahradil papirus, v stredoveku papier a knihy. Doba pokročila a dnes máme dáta digitalizované na magnetických a optických diskoch, prípadne FLASH pamätiach. A v neposlednom rade sú dnes dáta dostupné najmä zo siete internet.

Samotné informácie sú ale takmer k ničomu, ak sa v nich nebude dať nájsť hľadaný obsah. Možno aj pre túto skutočnosť sú dnes jedny z najpopulárnejších služieb na internete práve vyhľadávače. Za všetky spomeniem niekoľko najkvalitnejších – Google, Bing a Yahoo. O niektorých ich mňa-inšpirujúcich vlastnostiach napíšem niekoľko myšlienok ďalej v práci.

1.2 Potreba modernej vyhľadávacej služby pre IS Munis

IS Munis je informačný systém pre mestá a obce od firmy Triada spol. s r.o. V tejto spoločnosti popri štúdiu pracujem a podieľam sa na vývoji novej verzie tohto IS. So zvolením pána Mgr. Tomáša Lechnera, môjho nadriadeného, môžem použiť časť zdrojových kódov, ktoré som napísal ja k potrebám bakalárskej práce. IS Munis prechádza generačnou obmenou a snaží sa držať trend nastolený vo vyhľadávaní informácií. Pôvodný systém je dnes už nevyhovuje nastolenému trendu a vznikla potreba vytvoriť niečo nové, spĺňajúce potreby užívateľov IS Munis. Taktiež bol kladený dôraz na použitie aj v iných aplikáciach a nezávislosť na databázovom stroji.

1.3 Ciele knižnice

Hlavný cieľ tejto knižnice je poskytovanie zoznamu reálnych objektov z databáze na základe zadávaného slova. Strohé napovedanie ale nebolo pre celý IS dostatočné. Predstavme si modelovú situáciu, že užívateľ si pamätá telefónne číslo, ale nemôže si spomenúť na meno osoby, ktorej toto číslo patrí (stáva sa to aj mne, niekedy si číslo pohľadom zapamätám ale meno mi vypadne aj keď som sa ho snažil zapamätať). V tomto prípade samozrejme vyhľadávame majiteľa číslo, nie číslo samotné, keďže to je užívateľovi dobre známe. Hovoríme o tzv. kontexte. Kontext by mala byť jedna z predností tohto projektu.

Ďalší, nemenej podstatný cieľ je nezávislosť na databázovom stroji. V postate limit by mal byť len na platforme .NET a OleDbProvideri. Z praxe je známe, že podporované sú najpoužívanejšie databáze, ako Oracle, MS SQL, MySQL, PostgreSQL a taktiež napríklad MS Access.

Za zmienku určite stojí aj snaha o jednoduchú konfiguráciu, napríklad pomocou XML súboru, kde si namapujete Vašu databázu a určíte tabuľky a sloupce ktoré majú slúžiť na vyhľadávanie informácií v databáze. Rovnako by sa, napríklad v tomto súbore, mali dať našpecifikovať pokročilejšie voľby ako vlastná porovnávacia funkcia, ohľad na veľkosť písmen, alebo obmedzenie vstupu regulárnym výrazom a v neposlednom rade samotné kontexty.

V nasledujúcej kapitole sa pozrieme hlbšie do témy, čo všetko obnáša konfigurácia, ktoré údajne sú povinné, ktoré naopak nie. Preberieme možné prostriedky realizácie a povieme si niečo o API a vnútornej stavbe knižnice. V kapitole tiež načrtnem možné prístupy k problému podobnosti dvoch reťazcov.

2 Analýza

2.1 Očakávania používateľa

Z dlhoročných poznatkov firmy Triada spol. s r.o. vyplynulo, že typický užívateľ informačného systému Munis vyhľadáva informácie v systéme nasledovne: Pozeraním na klávesnicu napíše požadované kľúčové slovo a stlačí Enter, čím vyvolá prehľadávanie databázy na dané slovo. Ak neurobí typicky typografickú chybu, všetko by malo prebehnúť v poriadku a mal by sa dopracovať k výsledku. Ak však pri písaní spraví niekoľko chýb, alebo ani nevie čo presne hľadá, nastáva problém a vyhľadávač zväčša vráti prázdnu množinu výsledkov.

Z tohto plynie záver, že užívateľ očakáva interaktivitu a už po pár znakoch očakáva zobrazovať potenciálne hľadané objekty. Každým pridaným písmenom očakáva presnejšiu množinu výsledkov, až sa vo výbere objaví hľadaný objekt. V tomto momente už väčšina nebude text dopisovať, ale myšou vyberie objekt, ktorý si aplikácia môže následne vyžiadať z databázy.

Nie je preto prekvapením, že vývoj Munis Instant sa uberal práve týmto smerom. Priekopníkom v tejto technológii je firma Google, ktorá už podobný systém uplatňuje nejaký ten rok na svojom vyhľadávači. Inšpiráciu som čerpal práve na ich stránkach [1].

2.2 Kontext

Často sa užívateľ ocitne v situácii, že pozná informáciu, ktorá súvisí s hľadaným objektom, ale o tomto objekte nič viac nevie. Pripomeňme si modelovú situáciu zo strany 5. Poznáme telefónne číslo patriace osobe, ale zabudli sme/nepoznáme meno. V takomto prípade rozhodne nechceme, aby nám vyhľadávač ponúkol telefónne číslo, ktoré ak aj nepoznáme celé, príliš nás nezaujíma. To čo nás zaujíma je vlastník

čísla, teda osoba. V ponuke výsledkov teda užívateľa zaujíma meno osoby, ktorej telefónne číslo patri, druhotná informácia je samozrejme zadávané telefónne číslo, ktorá by sa tam mala rovnako zobraziť.

Čo ak je osoba vlastniaca telefónne číslo zamestnancom nejakej spoločnosti v databáze? V niektorých prípadoch nás môže zaujímať práve táto spoločnosť. Určite je preto viac než vhodné dovoliť našpecifikovať viac kontextov pre jednu hodnotu a dovoliť hĺbku zanorenia kontextov viac ako 1 (kontext kontextu).

2.3 Porovnávanie vstupu a hodnoty z databázy

Prvý z možných prístupov na porovnanie dvoch reťazcov je porovnať ich na čiastočnú zhodu. Môžme porovnávať na zhodu začiatku hodnoty z databázy k poskytnutému vstupu, prípadne môžme môžme tento výskyt hľadať kdekoľvek v slove z databázy. Toto riešenie má niekoľko výhod, ale aj nevýhod. Nesporná výhoda je, že tento prístup podporuje veľká väčšina databáz, v podstate všetky tie, ktoré majú implementovanú SQL konštrukciu LIKE. Pri použití indexovania dokáže byť aj dostatočne rýchly. Hodí sa porovnávanie číselných hodnôt, kde 123 nie je to isté ako 132. Naopak, vôbec nedokáže akceptovať preklepy do istej miery a tiež si nedokáže poradiť s diakritikou. Pokiaľ sú v databáze uložené hodnoty s diakritikou a na vstupe sa diakritika nepoužíva - veľmi časté - je v rozumnom čase nereálne zistiť ako by mal vyzerať tvar vstupu s použitou diakritikou.

Druhý prístup využíva procedurálne rozšírenie databázového stroja. Z toho teda plynie, že databázy bez možnosti tvorby užívateľsky definovaných funkcií budú o túto možnosť ukrátené. Jedná sa user-defined funkciu, ktorá ako vstup dostane 2 hodnoty a vzájomne ich porovná. Výstupom funkcie je reálne číslo [0;1] (uzavretý interval). Čím väčšie číslo, tým sú si hodnoty podobnejšie. Súčasťou takto definovanej funkcie môže byť aj možnosť odstránenia diakritiky obom parametrom, čím ich prevedieme do akéhosi normalizovaného tvaru. Na samotné porovnanie takto

normalizovaných hodnôt existuje niekoľko metód, viac k nim napíšem v ďalšej sekcii.

Ako ďalší prístup sa dá považovať využitie fulltextu poskytovaného databázovým strojom. Tento prístup sa javí ako najuniverzálnejší, avšak má niekoľko významných nedostatkov. Fulltext je riešenie na úrovni databázy a teda kód generovaný pre jednu databázu nie je funkčný na konkurenčnej technológii. Ďalší problém je, že nie všetky databázové systémy majú fulltext implementovaný. Pokiaľ implementovaný je, zvykne mat problémy s diakritikou a slovo bez diakritiky nie vždy dokáže spoľahlivo porovnať voči ekvivalentu s diakritikou. Taktiež sa väčšinou nedá nijakým spôsobom nastaviť citlivosť na preklepy a je otázne nakoľko je fulltext u jednotlivých databázových riešení preklepy rozoznávať.

2.4 Možné implementácie podobnostnej funkcie

Podobnosť dvoch textových reťazcov je celkom nejednoznačná vec. Každý človek posudzuje podobnosť subjektívne a neexistuje algoritmus, o ktorom by sme mohli bez že je najlepší pre všetky použitia. Existuje však niekoľko kvalitných algoritmov hodnotenia podobnosti reťazcov

2.4.1 Levenshteinova vzdialenosť

Levenshteinova vzdialenosť [2] je číslo udávajúce minimálny počet operácií takých, aby sme prvý reťazec pretransformovali na druhý (vytvorenie presnej kópie). Operácie sú 3 a to:

- substitúcia znaku za iný (vhodný)
- zmazanie znaku
- pridanie znaku

Čím je počet operácií vyšší, tým sú si reťazce menej podobné. Toto číslo následne

normalizuje na interval [0;1] aby spĺňalo návratovú hodnotu podobnostnej funkcie (tu už naopak väčšie číslo znamená väčšiu podobnosť).

2.4.2 Jaro-Winklerova vzdialenosť

Jarova vzdialenosť [3] $\mathbf{d_j}$ je definovaná nasledovne: $d_j = \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{(m-t)}{m}\right)$, kde **m** je počet zhodných znakov porovnávaných reťazcov **s1** a **s2**. Zhodné v tomto prípade znamená, že majú rovnakú hodnotu (**a,a** majú rovnakú hodnotu **a**, **a,b** naopak majú rozdielne hodnoty) a ich vzdialenosť v rámci reťazcov nie je väčšia ako $\left(\left(max(|s1|,|s2|)\right) \operatorname{div} 2\right) - 1$. **t** je 1/2 počtu transponovaných dvojíc znakov (častá typografická chyba, prehodenie dvoch znakov, napríklad "**lenght**" -> "**length**"). Prehodené znaky, ktoré sú od seba vzdialené o viac ako 1 znak sa za transpozíciu nepovažujú. Číslo $\mathbf{d_j}$ je z intervalu [0;1] (m je maximálne tak veľké ako dĺžka kratšieho z reťazcov), čo nám vyhovuje a výsledok nie je treba ďalej normalizovať.

Mierne upravená forma Jarovej vzdialenosti sa nazýva Jaro-Winklerova [4] vzdialenosť, ktorá zvýhodňuje reťazce s rovnakým začiatkom. Jaro-Winklerova vzdialenosť $\mathbf{d_w}$ je definovaná nasledovne: $d_w = d_j + (lp(1 - d_j))$, kde **l** je počet prvých spoločných znakov až do dĺžky 4. Ďalšie spoločné znaky už nehrajú pri zvýhodňovaní úlohu. **p** je konštanta, zvýhodňovací faktor. Jej veľkosť my nemala prekročiť 0.25, inak by $\mathbf{d_w}$ mohlo prekročiť hodnotu 1. Štandardne je táto hodnota nastavovaná na 0.1 .

2.4.3 Jaccardiho podobnostný index

Jaccardiho podobnostný index [5] je ďalšia možnosť ako vyjadriť podobnosť dvoch reťazcov. Je definovaný nasledovne: $J(A,B) = \frac{(A \cap B)}{(A \cup B)}$, kde **A** je množina znakov z prvého reťazca a **B** je množina znakov z druhého reťazca. 2 (a viac)

rovnaké znaky v jednom reťazci sa počíta ako 2 (a viac) prvkov množiny. Prienik množín je nanajvýš tak veľký ako ich zjednotenie, preto výsledný pomer je z intervalu [0;1].

2.4.4 Zhodnotenie

Levenshteinova vzdialenosť je pre naše potreby asi najmenej vhodná metóda. Vôbec si nerozumie s prehadzovaním susedných písmen, dokonca na jednu takúto opravu potrebuje 2 operácie a tým celkom znižuje rating vstupu.

Naopak zvyšné 2 prístupy sa javia ako veľmi použiteľné. Najlepšie porovnať reťazce s ohľadom na preklepy pravdepodobne dokáže Jaro-Winklerova vzdialenosť, berie ohľad na preklepy susedných znakov a tiež zvýhodňuje slová ktoré majú prvé znaky spoločné. Táto funkcia však musí bežať na databázovom stroji a jej časová zložitosť nie je úplne zanedbateľná, pretože vyžaduje analýzu poskytnutých reťazcov na zistenie počtu transpozícií. Pri veľkom množstve dát v databáze sa toto môže negatívne prejaviť na výkone.

Jaccardov index je v porovnaní s Jaro-Winklerom príliš benevolentný a v podstate slovo otočné voči inému má index podobnosti 1. Vychádza to z toho, že Jaccardov algoritmus nijak nesleduje poradie znakov. Teda aj transpozície nepovažuje za problém a rovnako majú takéto slová index podobnosti 1. V ľudskej reči a jazyku ale funguje tento algoritmus veľmi prijateľne. Výpočet pozostáva z aritmetických operácií a aj pri veľkom množstve dát bude pracovať plynule. Doložím v kapitole 4, kde sa budem zaoberať výkonom knižnice.

V mojej implementácii som použil algoritmus Jaccardovho indexu, práve preto, aby databázový stroj aj pri väčšom množstve dát a paralelnom spracovaní požiadavkov stále dosahoval vysokého výkonu. Samozrejme moje riešenie je skôr len príkladové, keďže ide o funkciu na úrovni databázy a vzhľadom na rozdielnosť rôznych DB

riešení ani nie je v mojich silách implementovať túto vlastnosť na všetky možné databázové systémy. Každý, kto bude chcieť tento projekt použiť si môže podobnostnú funkciu naimplementovať ako uzná za vhodné.

Určite je vhodné porovnávané reťazce pred samotným porovnaním znormalizovať (napríklad odstránením diakritiky a prevedením všetkých znakov na upper case). Zjednoduší sa tým vyhľadávanie pre užívateľov.

2.5 Konfigurácia

Konfigurácia by mala byť čitateľná a jednoduchá na pochopenie. Konfiguračné ini súbory už sú dnes menej použiteľné ako v minulosti. Naopak uplatnenie v tomto segmente stále viac nachádza technológia XML [6]. Prináša so sebou možnosť popisovať nastavenia a vlastnosti programu pomocou značiek. Tiež existujú možnosti ako predpisovať, ktoré značky a za akých okolností sa majú použiť a čo má byť ich obsahom. Platné v tomto smere sú technológie DTD [7] a XML Schema [8]. Platforma .NET cez natívne knižnice XML veľmi dobre podporuje a veľmi dobre sa s tým pracuje.

2.5.1 Nutné konfiguračné údaje

Prvé čo potrebujeme pre prístup do databázy, je poznať login, heslo, meno databázy a IP adresu databázového stroja. U SQL providerov platformy .NET je bežné tieto údaje uvádzať v tzv. Connection string-u. Pre programátora, ktorý bude eventuálne túto knižnicu používať je celkom jednoduché takýto connection string ku svojej databáze získať. Konfiguračný súbor preto na prístup do databáze akceptuje práve túto formu predávania prihlasovacích údajov. S prístupom do databáze súvisí aj podobnostná funkcia. Je potrebné našpecifikovať jej názov a index podobnosti, od ktorého sa výsledky začínajú počítať za relevantné.

Druhá časť konfiguračného súboru zahŕňa mapovanie tabuliek a ich stĺpcov do vyhľadávacích polí. Pre každú takúto tabuľku musíme nadefinovať jej názov a primárny kľúč, ktorý bude dôležitý pri pri predávaní návratovej hodnoty užívateľovi.

Každá tabuľka musí mapovať aspoň jeden stĺpec, v ktorom sa bude vyhľadávať. Niekedy však informácia z jedného stĺpca nie je dostatočne presná a preto musí existovať možnosť zahrnúť do návratovej hodnoty viac stĺpcov z jednej tabuľky, aby spolu vytvorili presnú informáciu. Samozrejme, že ich poradie by malo byť definovateľné. Ďalej je celkom dobrý nápad obmedziť hodnoty v stĺpci regulárnym výrazom. Ak by povedzme stĺpec obsahoval len číselné hodnoty, je nezmyselné hľadať zhodu s reťazcom, ktorý obsahuje aj iné znaky. Zrýchli sa tým vyhľadávanie, keďže ušetríme niekoľko dotazov do databázy. Stĺpec musí ďalej obsahovať svoj názov a rôzne prepínače ako možnosť stĺpec nezobraziť vo výpise, dôraz na veľkosť písmen, alebo možnosť použiť podobnostnú funkciu, alebo porovnávať predikátom LIKE (napríklad u číselných hodnôt je porovnávanie cez podobnostnú funkciu menej vhodné). Stĺpec tiež nesie informáciu o dátovom type uložených dát. Toto nastavenie je zatiaľ nevyužité a nahrádza ho regulárny výraz. V budúcnosti by ale mohlo toto nastavenie pomôcť zrýchľovať hľadanie pri neprítomnosti regulárneho výrazu.

Niekedy je však úplná informácia uložená naprieč tabuľkami. V takejto situáciu nachádza uplatnenie **kontext**. Pre každý vyhľadávací stĺpec môže byť definovaných 1 a viac kontextov, v závislosti na potrebách aplikácie. Kontext si potrebuje niesť niekoľko ďalších informácií, medzi iným primárny kľúč do kontextuálnej tabuľky, názov tejto tabuľky a cudzí kľúč do pôvodnej tabuľky, s ktorou sa bude kontextuálna spájať. Kontext samozrejme môže obsahovať vnorené kontexty a to do neobmedzenej hĺbky. Pri rozumnom použití ale určite nechceme ísť hlbšie ako 3 úrovne, inak sa informácia stáva veľmi obsiahla a knižnica stráca význam rýchleho napovedača. Kontext, podobne ako vyhľadávací stĺpec, môže pozostávať z informácie uloženej vo viacerých stĺpcov tabuľky. Aplikuje sa rovnaký prístup definovania spolu-zobrazených stĺpcov.

Kontext tiež nesie informáciu o type hľadaného objektu. Strohé ID by nám bolo zbytočné, keby sme nevedeli, v ktorej tabuľke sa tento objekt nachádza. Typ je v podstate len textový názov typu, ktorý si určí programátor, ktorý sa rozhodne túto knižnicu použiť. Tento názov typu sa bude vyskytovať v návratovej hodnote a aplikácia sa podľa neho môže rozhodovať ktoré funkcie pre výber objektu použije (príklad, aplikácia môže disponovať funkciami **Company ReturnCompanyById(int id)** a **Employee ReturnEmployeeById(int id)** a na základe typu našepkávaného objektu sa rozhodne, ktorú z týchto metód zavolá). Do budúcnosti by som chcel v tejto oblasti zapracovať na priamom predávaní typu objektu a následnej možnosti použiť generické metódy, bez nutnosti sa nejakým spôsobom rozhodovať na základe textového reťazcu.

2.5.2 Document Type Definition (DTD)

Pre dodržanie nastoleného formátu formátu XML súboru bohate postačuje DTD. Nemá síce podporu dátových typov ako XML Schema, ale pravdupovediac väčšina údajov sú aj tak stringové údaje, snáď len s výnimkou hodnoty podobnostného indexu, ktorý je reálne číslo z [0;1] a túto podmienku ľahko skontrolujem v kóde. Celé čísla sú ešte vyžadované u poradia výpisu stĺpcov, túto podmienku, ale tiež ľahko overím v kóde.

DTD dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT munisInstant (database,mapping)>

<!ELEMENT database ((connectionString),
(similarityFunction,precisionIndex)?)>

<!ELEMENT mapping (table)+>

<!ELEMENT connectionString (#PCDATA)>
```

```
<!ELEMENT similarityFunction (#PCDATA)>
<!ELEMENT precisionIndex (#PCDATA)> <!-- Must be real number from
0(included) to 1(included). Higher means, results will be more similar
to the pattern -->

<!ELEMENT table (column)+>

<!ATTLIST table
      name CDATA #REQUIRED
      id CDATA #REQUIRED>

<!ELEMENT column (regex?,withColumn*,(type|context+))>

<!ATTLIST column
      name CDATA #REQUIRED
      datatype (string|int|float|DateTime) #REQUIRED
      caseSensitive (yes|no) "yes"
      order CDATA #REQUIRED
      dontDisplay (yes|no) "no"
      useSimilarityFunction (yes|no) "no">

<!ELEMENT regex (#PCDATA)>

<!ELEMENT type (display?, returnType)>

<!ELEMENT display (#PCDATA) >
<!ELEMENT returnType (#PCDATA)>

<!ELEMENT context (contextColumn+,(context+|type)) >

<!ATTLIST context
      table CDATA #REQUIRED
      fk CDATA #REQUIRED
      id CDATA #REQUIRED>

<!ELEMENT contextColumn (withColumn*)>

<!ATTLIST contextColumn
      name CDATA #REQUIRED
      datatype (string|int|float|DateTime) #REQUIRED
```

```
        order CDATA #REQUIRED>

<!ELEMENT withColumn EMPTY>

<!ATTLIST withColumn
        name CDATA #REQUIRED
        datatype (string|int|float|DateTime) #REQUIRED
        order CDATA #REQUIRED>
```

Jediný element, ktorý nebol spomenutý v analýze vyššie, je element **display**, ktorý slúži na zobrazovanie pomocných informácií. Príklad: Zadávam nejaké číslo, knižnica nájde zhodu, priradí kontext a dá na výstup. Jediný problém ktorý nastal je, že neviem čo to číslo znamená. Jediné čo viem, že má nejaký súvis s kontextom. Element **display** so sebou nesie informáciu o tomto čísle, príklad, že sa jedná o telefónne číslo.

2.6 Návratová hodnota

Návratová hodnota nemôže byť len strohý text s názvom a krátkym popisom hľadaného objektu. Minimálne musí obsahovať identifikátor objektu v databázovej tabuľke a tiež akýsi typ objektu, ktorý jednoznačne pre aplikáciu identifikuje tabuľku, v ktorej sa má objekt nachádzať.

Zobrazovaný text by ale tiež nemal byť len statický textový reťazec, ale mal by poskytovať istú mieru interaktivity. To znamená, že užívateľ by rád vedel, ktorá časť zadávaného textu mu matchuje s ponúkaným výsledkom. Preto je dobré rozdeliť text na niekoľko častí a to **kontext**, **text pred v hľadaným reťazcom**, **matchujúca časť v hľadanom slove**, **zvyšok hľadaného slova**, **text po hľadanom slove**. Takto rozdelená informácia nám dovolí formátovaním textu oddeliť rôzne časti výstupného textu. Príklad: Lukáš Tomasy, tel: **733252**755. Výrazným fontom oddelená zadaná časť vzorku.

Návratová hodnota je zoznam objektov združujúcich položky popísané v predchádzajúcich odstavcoch. Teda takýto objekt musí obsahovať ID objektu k objektu, typ objektu (respektíve reťazec interne identifikujúci tabuľku, kde sa objekt nachádza) a formátovateľný text rozdelený do niekoľkých položiek. Toto by mali byt postačujúce položky a ďalšie informácie by boli už len doplnkové.

3 Prezentačná aplikácia s modulom Munis Instant

Každá knižnica potrebuje aplikáciu, ktorá dokáže naplno využiť jej potenciál. V rámci tejto práce som preto aj ja vytvoril aplikáciu, ktorá spôsobom návrhu môže pripomínať reálnu aplikáciu využívajúcu vlastnosti knižnice Munis Instant.

3.1 Popis aplikácie

Aplikácia, ktorú som stroho nazval Contact Manager je Client-Server aplikácia postavená na Microsoft WCF technológii [9]. Podporuje používanie viacerými používateľmi a pre každého ukladá vlastné dáta. Dáta spočívajú v ukladaní osôb a k nim ľubovoľné druhy kontaktov, ktoré si špecifikuje sám používateľ (napríklad e-mail, telefón, adresa, ICQ, jabber, ...). Aplikácia má povolené záznamy o osobách ľubovoľne meniť, dokonca je dovolené pridávať a odoberať druhy kontaktov. Manager má samozrejme vybavenie pre podporu Munis Instant, v pravej hornej oblasti sa nachádza vyhľadávacie pole, z ktorého sa po úspešnom vyhľadaní vyroluje zoznam výsledkov, a výberom jedného z nich vynútime detail osoby ktorú sme hľadali.

3.1.1 Server a jeho implementácia

Ako som už spomenul v odstavci vyššie, server je riešený ako WCF služba a je dostupný cez HTTP protokol na IIS serveri (Internet Information Services [10]). Štandardne je nastavený počúvať na porte 8081, ale toto sa dá v súbore Web.config zmeniť. IIS samozrejme musí mať vytvorenú asociáciu so službou a cestu k umiestneniu služby (štandardne v C:\BP_service, pokiaľ by bolo potrebné iné umiestnenie, je potreba prekompilovať zdrojové kódy so zmenenou konštantou homeDir na požadované umiestnenie. Je to pre podivné správanie IIS, ktorý za homeDir nepovažuje umiestnenie súborov služby, ale priečinok zanorený niekde

hlboko v C:\Windows. Dôvod pozná asi len Microsoft).

Ako databáza je použitý MS SQL server (v mojom prípade verzia 2008 Express), ale použitie iných databáz v podstate nie je problém, pokiaľ k nim existujú provideri a Connection string. MS SQL server má ale výhodu v podpore .NETu a umožňuje spúšťať funkcie napísané v C#. Celkom pohodlne sa dali naimplementovať funkcie pre použitie Munis Instantom.

Na perzistenciu objektov z databázy som použil open source riešenie NHibernate [11], ktoré dokáže dáta z relačnej databázy previesť do objektového modelu, s ktorým sa potom veľmi dobre ďalej pracuje. NHibernate mapuje dáta do predvytvorených tried pomocou mapovania popísaného v XML súboroch (ku každej triede jeden súbor). NHibernate podporuje tzv. lazy loading, čo znamená, že dáta sú loadované, až keď sú potrebné. Kolekcie objektov nižšie v hierarchii databáze sa inicializujú, až keď k ním reálne niekto pristúpi.

Vnútorne má server implementované metódy na prácu s perzistentnými objektami, na pridávanie objektov, ich mazanie a podobne. Takto upravené objekty je NHibernate opäť schopný uložiť do databáze, teda do relačných dát.

Rozhranie pre komunikáciu s klientom spočíva v sade metód, ktoré umožňujú vyššie popísanú funkcionalitu. Metódy, ktoré vracajú užívateľské dáta, ich poskytujú zapúzdrené v triede **User**, ktorá v sebe obsahuje informácie u aktuálnom userovi, jeho kontakty i typy spojení na ne. Pri práci sa zmeny prevádzajú priamo v tomto objekte a pri požiadavke užívateľa sa objekt pošle naspäť na server, kde metódy na to určené prevedú zmeny do databázy.

Okrem správy kontaktov, obsahuje server ešte metódu, ktorá sprístupňuje funkcie knižnice Munis Instant. Volá sa **Search** a môžete ju nájsť v Service Contracte

popísanom nižšie.

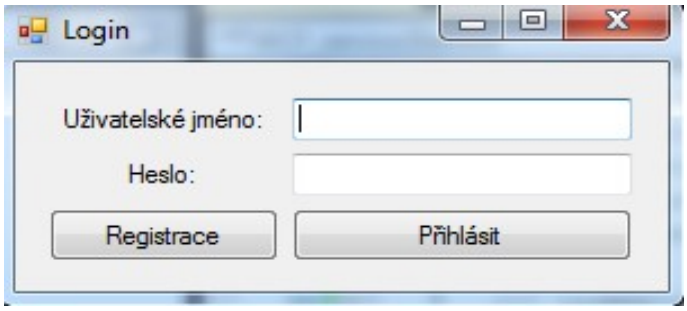
Service contract vyzerá nasledovne:

```
[ServiceContract]
public interface IService
{
    [UseNetDataContractSerializer]
    [OperationContract]
    User GetUserByID(int ID);

    [UseNetDataContractSerializer]
    [OperationContract]
    User GetUserByIDWithContact(int userID, int contactID);

    [UseNetDataContractSerializer]
    [OperationContract]
    User GetUserByLoginAndPwd(string login, string pwd);

    [UseNetDataContractSerializer]
    [OperationContract]
    User GetUserByCriteriaName(string Firtname, string Lastname, int UserID);

    [UseNetDataContractSerializer]
    [OperationContract]
    User GetUserByCriteriaType(int Type, string Context, int UserID);

    [UseNetDataContractSerializer]
    [OperationContract]
    bool SaveOrUpdateUser(User u);

    [UseNetDataContractSerializer]
    [OperationContract]
    bool RegisterNewUser(string login, string pwd);

    [UseNetDataContractSerializer]
    [OperationContract]
    IList<InstantObject> Search(string pattern, int userID);

    [UseNetDataContractSerializer]
    [OperationContract]
    bool DeleteContact(Contacts contact);

    void Dispose();
}
```

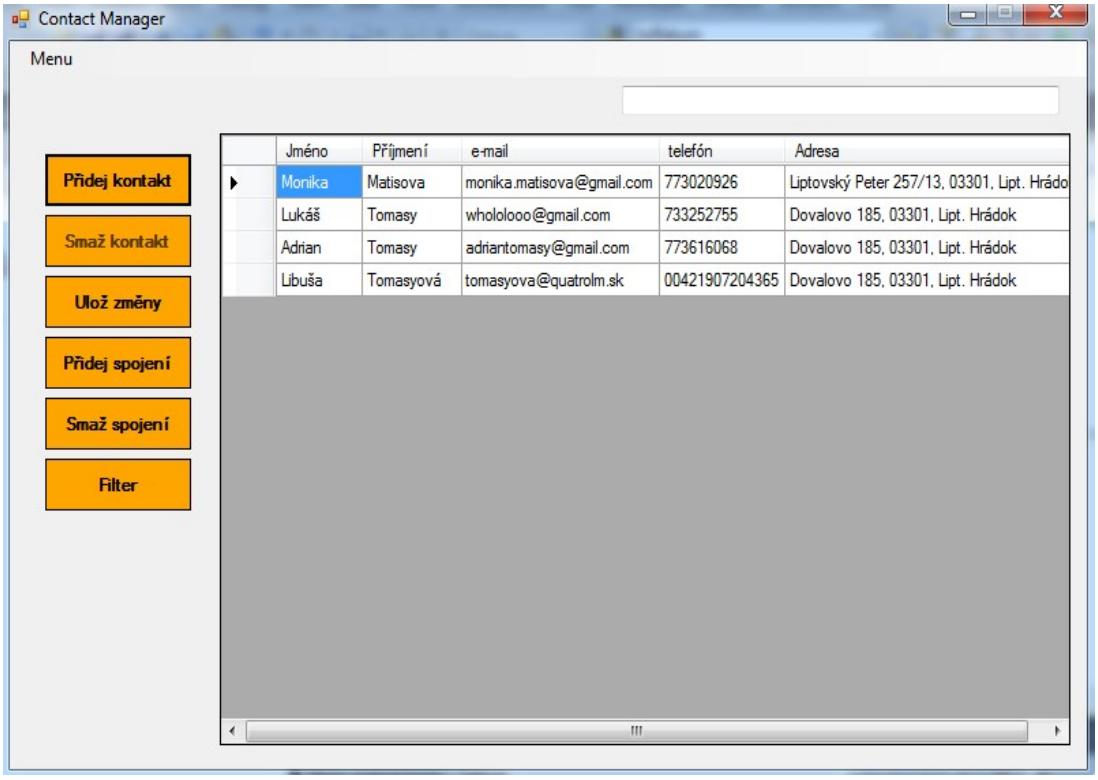
Popis metód kontraktu je možné nájsť v zdrojovom kóde v XML komentároch.

3.1.2 Klient a jeho GUI

Klient pozostáva s dvoch hlavných okien. Prihlasovacieho okna a hlavného okna. Prihlasovacie okno sa zobrazí okamžite po spustení aplikácie a umožňuje ako prihlásenie existujúceho používateľa, tak i registráciu nového. Po úspešnom prihlásení/registrácii, sa zobrazí hlavné okno so zoznamom kontaktov a panelom nástrojov na strane ľavej.



prihlasovacie okno

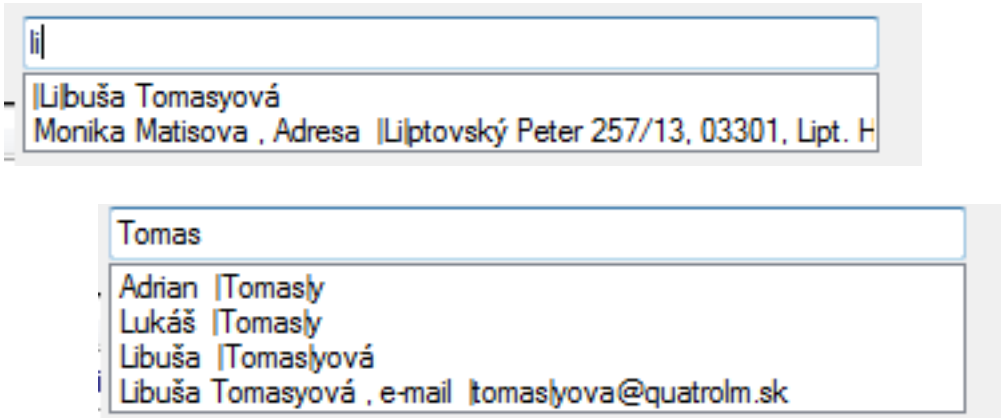


hlavné okno

Nástrojmi na ľavom paneli, sa dá pracovať s adresárom. Tlačítkom **Přidej kontakt**

sa vyvolá formulár na pridanie novej osoby do adresáru. Tlačítkom **Smaž kontakt** naopak zmažeme vybraný kontakt (pri nevybraní žiadneho kontaktu, je tlačítko neaktívne). Stlačenie tlačítka **Ulož změny** sa prevedené zmeny uložia do databázy. Zmeny sa neukladajú automaticky pri zmene hodnoty. V prípade že si zmeny uložiť zabudnete, program vás na to automaticky upozorní. Tlačítka **Přidej spojení** a **Smaž spojení** vyvolávajú formuláre na pridanie/odobranie spojenia všetkým kontaktom. Nakoniec tlačítko **Filter** filtruje kontakty podľa rôznych kritérií, ale pracuje s lokálnymi dátami, nie s dátami v databáze.

S touto bakalárskou prácou má súvis ale len pole pre vkladanie text v pravom hornom okraji. To je miesto, odkiaľ sa vyvoláva hľadanie pomocou Munis Instant. Je nastavené tak, že každým znakom sa vyvolá hľadanie daný reťazec. Pre výber jedného z výsledkov hľadania, je potrebné naň kliknúť myšou, pripadne stlačením kurzorovej šipky nadol vyvolať vyberanie šípkami nahor a nadol. Enterom potom potvrdíme výber.



Munis Instant v praxi

Bohužiaľ, moje designerske schopnosti nie sú na vysokej úrovni, aplikácia sa preto môže používať ťažkopádnejšie. Za to sa samozrejme ospravedlňujem, ale ako som spomenul vyššie, slúži primárne na prezentovanie schopností knižnice Munis Instant v prostredí podobnom reálnemu nasadeniu.

4 Implementácia a optimalizácia Munis Instant

4.1 Implementácia

4.1.1 Implementácia knižnice

Základnou triedou celej knižnice je trieda MunisInstant, ktorá implementuje interface IMunisInstant. Konštruktor je buď bezparametrický, alebo s parametrom, ktorý udáva cestu k homeDiru. Pri vytvorení objektu pomocou bezparametrického konštruktoru sa za homeDir považuje **./** , čo vo väčšine prípadov je adresár, kde sa nachádza MunisInstant.dll (výnimku tvorí IIS server, ktorý ako som popísal v texte vyššie má trošku divné správanie v tomto smere). Konštruktor ďalej volá funkcie na validáciu XML konfiguračného súboru, ktorý hľadá v homeDire. Porovnáva ho k DTD súboru, taktiež umiestnenom v homeDire, prípadne homeDir\bin . Pokiaľ sa validácia nepodarí, dôjde k chybe, ktorá bude zaznamenaná v logu.

V prípade, že sa validácia podarí, nasleduje parsovanie XML dokumentu a prevádzanie údajov do objektov T**able** pre každý element table, C**olumn** pre každý element column, **Context** pre každý element context. Tieto triedy berú v konštruktore za parameter **XElement element**. Parsovaním XML dokumentu teda vzniká stromová štruktúra objektov. Pri parsovaní môže opäť dôjsť k chybám, napríklad keď sa nepodarí previesť prečítanú hodnotu na číslo. O ostatné by sa už malo postarať DTD a jeho validácia. Problémy mapovania tabuliek a stĺpcov sa odhalia až za behu programu, keď vznikne SQL dotaz, s neplatnými hodnotami.

Ak všetko prebehlo v poriadku, objekt je pripravený na použitie. Zavolaním metódy Search(pattern), sa vyvolá postupnosť vytvárania SQL dotazov. Každá namapovaná tabuľka si vyžiada od svojich vyhľadávacích stĺpcov dotazy, ktoré následne zlepí do zoznamu. Column si už dotaz vie poskladať takmer celý, má k tomu všetky

informácie čo potrebuje. Vie, či má byť použitá podobnostná funkcia, či má byť hľadanie case sensitive a taktiež pozná všetky stĺpce, ktoré majú byť v dotaze uvedené. Jediné čo nepozná, je kontext. Kontext totiž môže mať hĺbku zanorenia kontextov väčšiu ako 1. Tento problém som vyriešil hniezdenými dotazmi. Column pozná svoje Contexty, preto stačí aby Context vedel takýto hniezdený dotaz vygenerovať. Takto vygenerovaný dotaz sa zabalí s ďalšími servisnými dátami (pozícia kontext, pozícia matchujúceho slova, ...) zabalí do objektu a vráti sa ako návratová hodnota.

Keď už sú všetky dotazy vygenerované, začíname ich servírovať databázovému stroju. Práve tu sa môžu vyskytnúť SQL výnimky, na ktoré ktoré knižnica skončí s chybou. Chyba sa samozrejme zapíše do logu. Každá odpoveď od databázy sa následne pomocou servisných dát spracuje na zoznam objektov obsahujúci ID objektu, typ objektu, kontext, text pred písaným slovom, napísaná časť písaného slova, zvyšok tohto slova a nakoniec text po písanom slove. Zoznamy z každého dotazu sa zreťazia do jedného zoznamu a posielajú sa ako výsledok na klienta.

4.1.2 Implementácia podobnostnej funkcie

Podobnostnú funkciu som implementoval pre MS SQL Server 2008, ktorý umožňuje spúšťať .NETovský kód. Aplikoval som Jaccardov index, ktorý je na výpočet časovo príjemne nenáročný a aj pri väčšom objeme dát a vyššom vyťažení databázového stroja nebude príliš brzdiť vyhľadávanie. Testy na vzorke cca 6250 záznamov českých miest a obcí doložím v závere tejto kapitoly.

Pre špecifické potreby knižnice som Jaccardov algoritmus mierne upravil. Užívateľ očakáva poskytovanie výsledkov už pri stlačení prvých znakov. Tu by samotný algoritmus nefungoval podľa predstáv a začal by poskytovať výsledky až pri dostatočne dlhých vstupoch. V podstate až vtedy, keď by bol za vstup poskytnutý reťazec, ktorý by bol ako prvý podobný hodnote v databáze. Toto chovanie ale nie

úplne vyhovuje požiadavkám používateľov. Preto som algoritmus upravil nasledovne. Jaccardov index spočítam 2x. Raz pre vstup a hodnotu z databázy, druhý krát pre vstup a prefix hodnoty z databázy rovnakej dĺžky ako je vstup. V prípade, že je vstup dlhší ako hodnota z databázy, samozrejme postačí Jaccardov index spočítať len raz. Prioritne sa za podobnosť považuje Jaccardov index spočítaný z celej hodnoty z databázy, avšak ak je rozdiel týchto dvoch výpočtov väčší ako zvolená konštanta (ja som zvolil konštantu 0.2) popisujúca dôležitosť prefixu voči celej hodnote, ako podobnosť sa zvolí Jaccardov index prefixu hodnoty z databázy a vstupu. Typicky takáto situácia nastane pokiaľ prefix matchuje vstup, zatiaľ čo celá hodnota je ešte príliš odlišná (pre Jaccardov index) voči vstupu, že algoritmus rozhodne o nepodobnosti.

Vďaka tomuto prístupu začne knižnica poskytovať výsledky už od prvého písmena, hoci veľmi nepresné, pretože sa v databáze môže potencionálne nachádzať veľké množstvo dát splňujúcich podmienky podobnostnej funkcie. Každým ďalším znakom sa ale výsledok spresňuje a pri dostatočne dlhom slove sa už porovnáva na celú dĺžku hodnoty, nie na jej prefix. Tu už je výsledok dostatočne presný.

4.1.3 Bezpečnosť

Aplikácie pracujúce s dátami pravidelne kladú veľký dôraz na bezpečnosť dát. Samozrejme je ochrana dát i v záujme knižnice Munis Instant, hoci dáta z databáze len číta, nezapisuje.

Najčastejšia forma útoku na databázové stroje je tzv. SQL Injection [12], teda preniknutie do databázy podvrhnutím vstupu, ktorý databázový systém vyhodnotí ako SQL príkaz. Takto môže útočník dáta poškodiť, ukradnúť, alebo aj nenávratne vymazať. Tento problém samozrejme knižnica Munis Instant rieši. Možných riešení je niekoľko, ja som zvolil elegantné riešenie, ktoré .NET OleDbProvider poskytuje a je ním použitie parametrov. Text získaný zo vstupu je uložený do parametru, ktorý je

následne vložený na predurčené miesto v SQL dotaze generovanom v kóde. Takto interpretovaný vstup sa pre SQL server správa ako čistý text, teda aj znaky ako **'** a **;** a ďalšie. Týmto sa zamedzí ohrozeniu dát metódou SQL Injection.

Zraniteľným miestom môže byť aj samotný Connection String uložený v konfiguračnom súbore. Obsahuje totiž prihlasovacie údaje do databázy. Použitie knižnice predpokladá nasadenie najmä na client-sever aplikáciach, hoci nasadenie na iných architektúrach aplikácií nie je vylúčené. Ak by bol tento konfiguračný súbor uložený niekde na serveri, pravdepodobne by k žiadnym komplikáciám nedošlo, keďže súbory na serveri uložené by mali byť zabezpečené proti neoprávnenému použitiu. Problém nastáva ak by mal typický užívateľ prístup k takto citlivým informáciám. Na tento problém však pravdepodobne neexistuje jednoznačný liek. Pevne verím, že administrátor aplikácie využívajúcej túto knižnicu si dá na svoje súbory veľký pozor.

4.1.4 Optimalizácia

Jeden spôsob optimalizácie som spomenul už v predchádzajúcich kapitolách. Ide o obmedzenie vstupu regulárnym výrazom. Nastavuje sa v konfiguračnom súbore značkou **regex**. Obmedzenie regulárnym výrazom spôsobí, že pre vstup, ktorý s ním nebude matchovať, nebude vytvárať SQL dotaz pre stĺpec, s ktorým je regulárny výraz zviazaný. Takto sa ušetrí čas vyhľadávaním v tabuľkách, kde by sa nemali vyskytovať hľadané informácie.

Druhý spôsob spočíva v použití cache pri používaní Munis Instant. Hľadané výsledky sa zachovávajú a pri odmazávaní znakov (backspace) sa neprehľadáva databáza znova, ale použije sa cache. Výrazne to urýchľuje hľadanie v prípade že si užívateľ uvedomil chybu a snaží sa slovo opraviť.

Pre čo najvyššiu efektivitu je takisto obmedzený počet položiek v odpovedi. Ten je

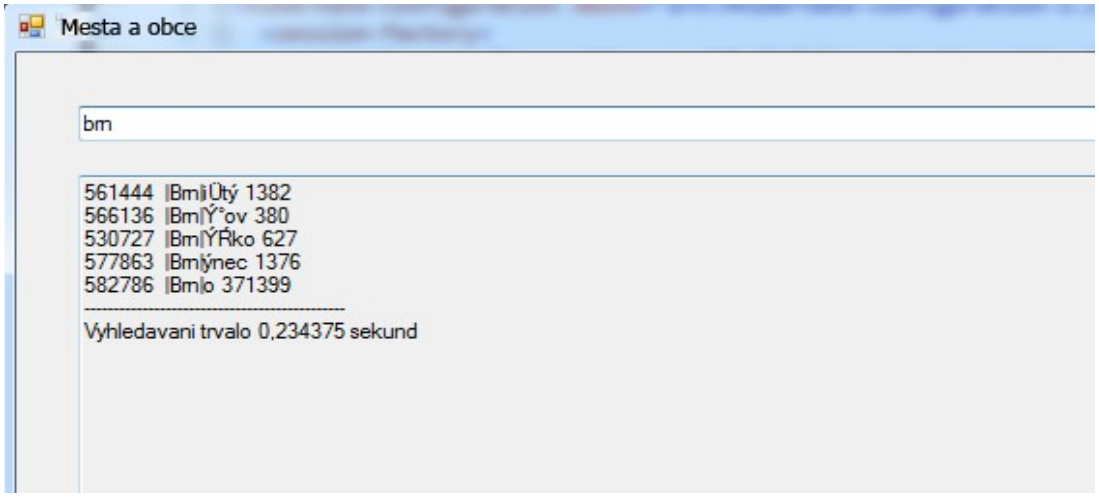
síce premenlivý, dokonca nemá ani nijak stanovené maximum, ale na jeden SQL dotaz je vrátených vždy maximálne 10 najvyššie hodnotených záznamov. Počet záznamov v odpovedi teda priamo závisí na počte generovaných dotazov. Tento aspekt má za úlohu aj udržiavať prehľadnosť odpovedí, vo veľkom množstve záznamov sa už rozhodne nedá hovoriť o rýchlom hľadaní.

4.1.5 Výkon

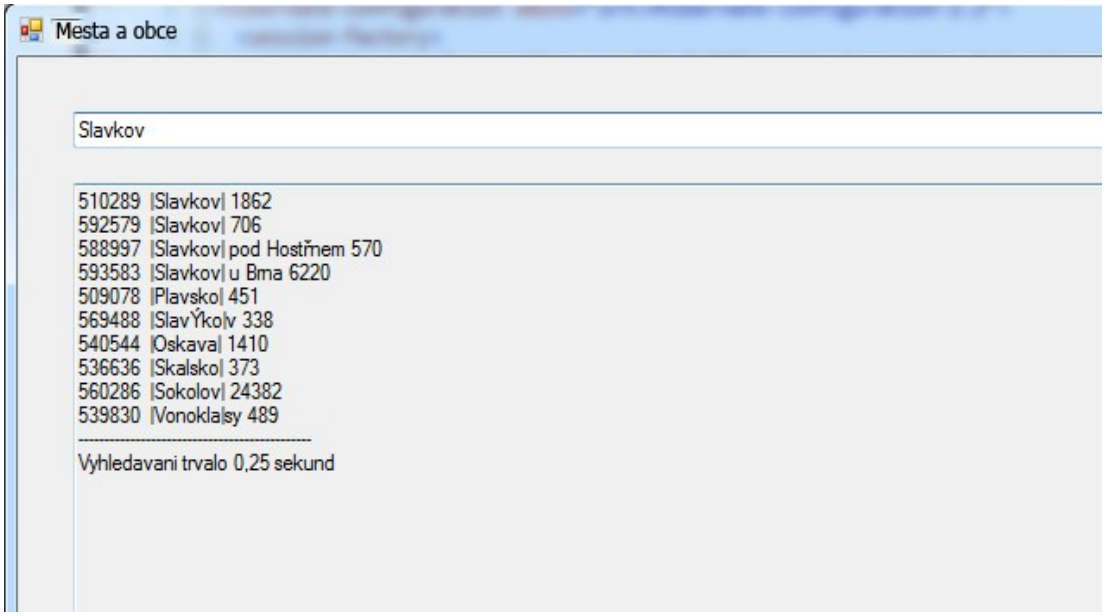
Na odmeranie výkonu vyhľadávania bola potrebná reprezentatívna vzorka. K dispozícii som mal zoznam cca 6250 českých miest a obcí aj s údajom o počte obyvateľov. Údaje boli uložené v CSV súbore a bohužiaľ, jeho kódovanie textu bolo odlišné od kódovania databázy. Diakritické znaky sú preto zdeformované. Pôvodne som chcel tento problém nejako napraviť, ale nakoniec som sa priklonil k tomu, že sa na takýchto dátach bude dať celkom dobre testovať podobnosť reťazcov, v ktorých sú chyby. Vyhľadávanie v tomto množstve dát spravidla trvalo okolo 0.3 sekundy, čo z pohľadu užívateľa môže byť celkom prijateľný čas.

Pri vyhľadávaní, ktoré nevyužíva podobnostnú funkciu je možné zrýchliť vyhľadávanie indexovaním podľa vyhľadávacieho stĺpca. Pri využití podobnostnej funkcie bohužiaľ táto možnosť nie je k dispozícii, potrebovali by sme totiž poznať hodnotu funkcie pre všetky možné vstupy, aby sme k nim vytvorili rýchle indexy. Počet možných vstupov z klávesnice je však potencionálne nekonečný.

Niekoľko meraní:



Hľadanie mesta Brno, 0.23 sekundy



Slavkov u Brna, 6220 obyvateľov

4.2 Inštalácia

4.2.1 Inštalácia Munis Instant

Samotný Munis Instant ako knižnica sa nijak neinštaluje. Pozostáva z troch súborov:

- MunisInstant.dll

    - binárny súbor knižnice

- MunisInstant.dtd

    - DTD schéma pre obmedzenie konfiguračného súboru len platnými tagmi

- instant.cfg.xml

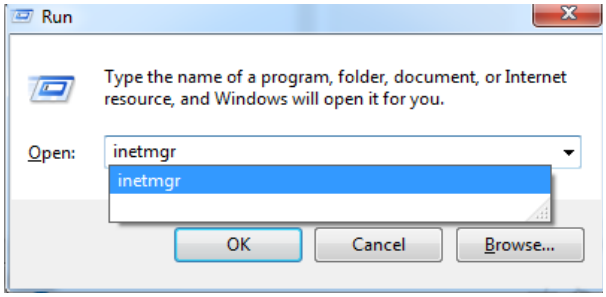
    - konfiguračný súbor pre knižnicu, mapovanie tabuliek a nastavovanie prepínačov

Knižnica sa používa ako každá iná, do projektu s aplikáciou si nareferencujeme dll súbor a vytvoríme inštanciu triedy MunisInstant implementujúcu interface IMunisInstant. Pri vytvorení objektu pomocou bezparametrického konštruktoru sa DTD a XML súbor hľadajú v priečinku s DLL súborom, naopak pri použití parametru **homeDir** sa DTD a XML súbor hľadajú v zvolenom priečinku.

4.2.2 Inštalácia Contact Manageru - server

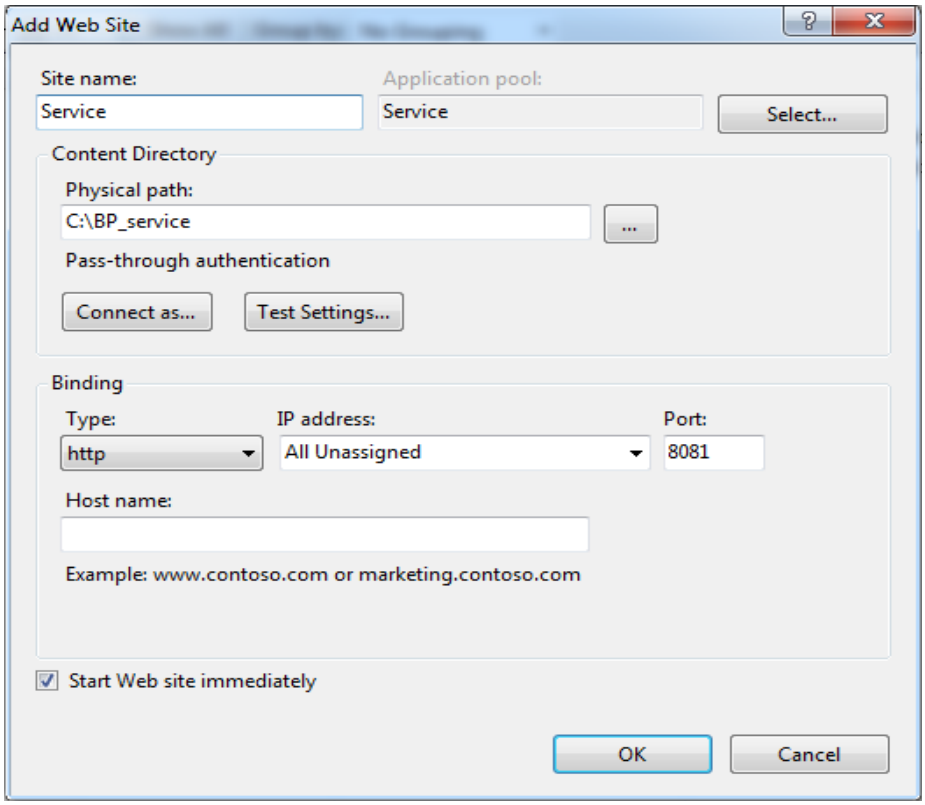
Pre inštaláciu tejto služby je potrebné mať niekoľko vecí. Na hostovanie služby potrebujeme IIS server, ktorý je ale dostupný bežne vo Windows, len zvyčajne nie je povolený. Prvý krok je teda uvedenie IIS do chodu. Následne je treba vytvoriť novú "stránku". Nie je to stránka ako ju poznáme z webu, ale ISS dokáže spoľahlivo spúšťať .NET kód a teda nielen ASP.NET stránky, ale aj WCF služby. Službu nabindujeme na požadovaný adresár (defaultne C:\BP_service , ale je možná zmena, avšak je potrebné prekompilovať zdrojové kódy s upravenou konštantou **homeDir** - pre podivné správanie IIS, ktoré som už v práci spomínal) a požadovaný port (defaultne 8081, je ale možné zmeniť, vyžaduje však zmenu portu v konfiguračnom súbore klienta). Následne to tohto adresára nakopírujeme súbory serveru. Na záver sa uistíme, či naša služba na IIS je spustená.

Pre lepšiu ilustráciu prikladám vytvorenie služby aj v obrazovej podobe:



Spustíme IIS Manager

Na ľavej lište vyberieme z kontextovej ponuky položky Sites **Add Web Site**



Vyplníme údaje **Site name, Physical path** a **Port**

Druhý krok je inštalácia .NET verzie 4.0. Pokiaľ ho nemáte, stiahnite si inštalátor zo stránok Microsoftu, je zdarma a nainštalujte.

Tretí krok je príprava databáze. Pokiaľ databázu nemáte pripravenú, urobte tak. Následne si vytvorte **Connection String**, napríklad pomocou Visual Studia. Tento Connection String použijete v súboroch instant.cfg.xml (konfiguračný súbor pre Munis Instant) a hibernate.cfg.xml (konfiguračný súbor pre NHibernate). U instant.cfg.xml je v **Connection Stringu** nutné navyše uviesť **Provider=SQLOLEDB** inak knižnica zahlási chybu.

4.2.3 Inštalácia Contact Manageru - klient

Klienta sa spúšťa súborom **WCF_client.exe**. V súbore **WCF_client.exe.config** je potrebné nastaviť niekoľko vecí. Tento súbor je XML súbor a nás bude zaujímať značka **endpoint** a jej atribút **address**, ktorý musíme nastaviť podľa umiestnenia serveru. Protokol je vždy HTTP, ten sa teda nemení. Mení sa len adresa a port. Nastavíme teda adresu stroja, kde služba beží a port na ktorom počúva.

Viac konfigurovať netreba, môžete spustiť.

5 Záver

Knižnica plní stanovené ciele v kapitole 1. Poskytuje vyhľadávanie v reálnom čase a to nielen napovedaním písaného slova, ale aj poskytovaním prípadného kontextu, aby informácia dávala význam.

Nezávislosť na databázovom stroji je podľa možností zaručená. Spoliehať sa treba podporu databázy OleDbProviderom. Základná funkcionalita je dostupná všetkým takýmto databázam, pretože dotazy sú tvorené len čistým SQL jazykom. Pokročilejšie funkcie knižnice sú už ale závislé na použitom databázovom stroji. Funkcie ako UPPER() / LOWER() nie sú v špecifikácii SQL jazyka, ich podpora naprieč databázami je však takmer 100%ná. Horšie to je ale s podobnostnou funkciou, ktorá využíva programovateľné rozšírenia databázových riešení.

Konfiguráciu som sa snažil urobiť čo najjednoduchšiu, zvolil som na to XML formát a všetky nastavenia sa nachádzajú v jednom súbore. Taktiež poskytuje DTD dokument, podľa ktorého si užívateľ môže svoj konfiguračný súbor zvalidovať. Na internete existuje niekoľko kvalitných XML editorov.

5.1 Rozšírenia do budúcnosti

Do budúcnosti ma napadá niekoľko užitočných rozšírení. Určite by bolo vhodné definovať aj viac ako len jednu podobnostnú funkciu. Každá sa totiž môže hodiť na iný typ dát. V tomto stave knižnica umožňuje len jednu takúto funkciu.

Ďalšie zlepšenie som už v texte načrtol a je ním genericita, s ktorou by sa mohlo pracovať pohodlnejšie ako s aktuálnym formátom poskytovania typu objektu. Uľahčilo by to písanie kódu a nahradilo viacnásobné vetvenie na základe poskytnutého typu.

Niektorým môže imponovať aj prípadná škálovateľnosť počtu záznamov v odpovedi. Zatiaľ je stanovená na pevnú hodnotu max 10 záznamov na SQL dotaz.

Literatúra

[1] Google Inc.: Google Instant,
http://www.google.com

[2] Levenshtein distance,
http://en.wikipedia.org/wiki/Levenshtein_distance

[3] Jaro distance,
http://en.wikipedia.org/wiki/Jaro–Winkler_distance

[4] Jaro-Winkled distance,
http://en.wikipedia.org/wiki/Jaro–Winkler_distance

[5] Jaccard index,
http://en.wikipedia.org/wiki/Jaccard_index

[6] XML technology,
http://en.wikipedia.org/wiki/XML

[7] Document Type Definition,
http://en.wikipedia.org/wiki/Document_Type_Definition

[8] XML Schema,
http://en.wikipedia.org/wiki/XML_Schema_(W3C)

[9] Micorosoft WCF,
http://msdn.microsoft.com/en-us/netframework/aa663324

[10] IIS - Internet Information Services,
http://en.wikipedia.org/wiki/Internet_Information_Services

[11] NHibernate,
http://nhforge.org/Default.aspx

[12] SQL Injection,
http://en.wikipedia.org/wiki/SQL_injection

Príloha A

Obsah CD

Súčasťou práce je aj CD-ROM so zdrojovými kódmi aplikácie a elektronickou podobou textu bakalárskej práce.